

# Programação Orientada a Objetos

- Classes de Objetos
- Construtor e Destruitor
- Ponteiros para Objetos
- Sobrecarga de Operadores
- Membros Estáticos
- Funções e Classes Amigas
- Herança de Classes
- Métodos Virtuais e Classes Abstratas
- Templates de Classes

} 2<sup>a</sup> Aula

# Programação Orientada a Objetos

- Classes de Objetos
- Construtor e Destruitor
- Ponteiros para Objetos
- **Sobrecarga de Operadores**

# Sobrecarga de Operadores

- A linguagem C++ permite a sobrecarga dos operadores *standard* na definição de classes definidas pelo programador e assim estender o seu funcionamento

```
= + - * / % & | ^ < > == != <= >= && ||
+= -= *= /= %= <<= >>= &= |= ^=
++ -- ! ~ [] () << >> new delete
```

- Para isso, basta declarar o operador na definição da classe e definir o método respetivo:

```
<tipoResultado> operator <sinalOperador> (<parâmetros>);
```

# Sobrecarga de Operadores

```
class CRetangulo{
    int *pAltura, *pLargura;
public:
    ...
    CRetangulo& operator = (const CRetangulo &); // existe por defeito...
    CRetangulo operator + (const CRetangulo &);
    CRetangulo& operator ++ (void); // operador "prefix"
    CRetangulo operator ++ (int); // operador "postfix"
};

CRetangulo& CRetangulo :: operator = (const CRetangulo &um) {
    *pAltura = *(um.pAltura); *pLargura = *(um.pLargura);
    return *this;
}
...
```

- Sobrecarga do operador =
  - Importante quando a estrutura de dados do objeto usa alocação dinâmica de memória: permite fazer uma “deep copy” dos atributos do objeto
  - Devolução de uma referência permite fazer atribuições em cascata, e.g. `a = b = c`

# Sobrecarga de Operadores

```

...
CRetangulo CRetangulo :: operator + (const CRetangulo &b) {
    CRetangulo r; // objeto auxiliar para calcular e devolver o resultado
    *(r.pAltura) = *pAltura + *(b.pAltura);
    *(r.pLargura) = *pLargura + *(b.pLargura);
    return r;
}

CRetangulo& CRetangulo :: operator ++ (void) {           // operador "prefix"
    *pAltura = *pAltura + 1; *pLargura = *pLargura + 1;
    return *this;
}

CRetangulo CRetangulo :: operator ++ (int) {             // operador "postfix"
// parâmetro "dummy": apenas para distinguir do método "prefix"
    CRetangulo ret_anterior(*this); // cópia p/ depois devolver
    *pAltura = *pAltura + 1; *pLargura = *pLargura + 1;
    return ret_anterior;
}
...

```

# Sobrecarga de Operadores

```

...
int main(void) {
    CRetangulo a(3, 7), b(6, 3), c;
    CRetangulo d, e, f(a), g(b), h, i;

    c = d = a;
    // também dá c.operator=(d.operator=(a))

    e = a + b;
    // também dá e = a.operator+(b); -> usa o operador =
    // portanto também dá: e.operator=( a.operator+(b) )

    h = f++;      // também dá: h.operator=( f.operator++(0) );

    i = ++(g); // também dá: i.operator=( ( g.operator++() ).operator++() );

    cout << "Áreas de a,b,c,d,e,f,g,h,i:" << endl << a.area() << ' '
        << b.area() << ' ' << c.area() << ' ' << d.area() << ' ' << e.area()
        << ' '<< f.area() << ' '<< g.area() << ' '<< h.area() << ' '<< i.area() << ' ' << endl;
}


```

Áreas de a,b,c,d,e,f,g,h,i:  
21 18 21 21 90 32 40 21 40

# Exemplo – Números Complexos

```

class CNumComplexo {
    double r, i;
public:
    CNumComplexo(void) { };
    CNumComplexo(double, double);
    void mostra(void);
    CNumComplexo operator * (const CNumComplexo &);
    void operator += (const CNumComplexo &);
};

CNumComplexo :: CNumComplexo(double a, double b) { r = a; i = b; }

void CNumComplexo :: mostra(void) {
    if (i > 0.0) cout << r << '+' << i << 'j' << endl;
    else if (i < 0.0) cout << r << '-' << -i << 'j' << endl;
    else cout << r << endl;
}

...

```

# Exemplo – Números Complexos

```

...
CNumComplexo CNumComplexo :: operator * (const CNumComplexo &b) {
    CNumComplexo prod(this->r*b.r-this->i*b.i, this->i*b.r+this->r*b.i);
    return prod;
}

void CNumComplexo :: operator += (const CNumComplexo &b)
{ this->r += b.r; this->i += b.i; }

int main() {
    CNumComplexo a(3, 1), b(1,-2), c, d;
    c = a * b;
    c.mostra();
    d = a;
    d.mostra();
    d += b;
    d.mostra();

    return 0;
}

```

5 - 5j  
3 + 1j  
4 - 1j

## Exemplo – Números Fracionários

```

class CFracao{
    int numerador;
    int denominador;
public:
    CFracao();           // inicialização por defeito
    CFracao (int, int);
    CFracao (const CFracao &); // E se não fosse definido?
    bool eNumeroInteiro();
    bool operator == (const CFracao &);
    CFracao operator + (const CFracao &);
};

CFracao :: CFracao() { numerador=denominador=1; } // garante denom.<>0

CFracao :: CFracao(int n, int d) { numerador=n; denominador=d; }

CFracao :: CFracao(const CFracao &uma) {
    this->numerador=uma.numerador; this->denominador=uma.denominador;
}
...

```

## Exemplo – Números Fracionários

```

...
bool CFracao :: eNumeroInteiro() {
    if (denominador == 0) return false;
    if (numerador % denominador == 0) return true; else return false;
}

bool CFracao :: operator == (const CFracao &uma) {
// se a/b == c/d então ad == bc
    if (this->numerador*uma.denominador==this->denominador*uma.numerador)
        return true;
    else return false;
}

CFracao CFracao :: operator + (const CFracao &b) {
// a/b + c/d = (ad+cb)/bd
    CFracao soma(numerador*b.denominador+b.numerador*denominador,
                 denominador * b.denominador);
    return soma;
}

```

## Exemplo – Polinómios 3º Grau

- Classe para representar polinómios do tipo  $a_3x^3+a_2x^2+a_1x+a_0$

```
class CPolinomio {
    double coeficiente[4]; // coeficientes: [0]=a0,[1]=a1,[2]=a2,[3]=a3
    int grau; // contém sempre o grau do polinómio
    int determinaGrau(void); // em função dos coeficientes
public:
    CPolinomio (); // inicialização por defeito: polinómio = 1
    CPolinomio(double, double, double, double); // inicializ. 4 coeficientes
    CPolinomio (const CPolinomio &); // inicialização por cópia

    int getGrau(void); // devolve o grau do polinómio
    CPolinomio derivada(void); // derivada do polinómio
    bool operator > (const CPolinomio &);
    CPolinomio operator + (const CPolinomio &);
};
```

## Exemplo – Polinómios 3º Grau

```
CPolinomio :: CPolinomio () {
    // inicialização por defeito: polinómio = 1
    coeficiente[0] = 1.0;
    for (int i = 1; i<=3; i++) coeficiente[i] = 0.0;
    grau = 0;
}

int CPolinomio :: determinaGrau (void) {
    for (int i = 3; i > 0; i--)
        if (coeficiente[i] != 0.0) return i;
    return 0;
}

CPolinomio :: CPolinomio (double a3, double a2, double a1, double a0)
{
    coeficiente[3] = a3; coeficiente[2] = a2;
    coeficiente[1] = a1; coeficiente[0] = a0;
    grau = determinaGrau();
}
```

## Exemplo – Polinómios 3º Grau

```
CPolinomio :: CPolinomio (const CPolinomio &p) {
    // construtor por cópia: e se não definissemos este método explicita/?
    for (int i = 0; i < 4; i++) coeficiente[i] = p.coeficiente[i];
    grau = p.grau;
}

int CPolinomio :: getGrau (void) {return grau;}

CPolinomio CPolinomio :: derivada(void) {
    // devolve o polinómio 3*a3*x^2+2*a2*x+a1
    CPolinomio p;

    p.coeficiente[3] = 0.0; // o coeficiente a3 é sempre zero
    for (int i = 3; i > 0; i--) p.coeficiente[i-1] = i*coeficiente[i];
    p.grau = p.determinaGrau();

    return p;
}
```

## Exemplo – Polinómios 3º Grau

```
bool CPolinomio :: operator > (const CPolinomio &p) {
    //compara dois polinómios: o primeiro é maior do que o segundo se:
    // - tem maior grau;
    // - com o mesmo grau, se os coeficientes de maior grau são maiores.
    if (grau > p.grau) return true;
    else if (grau < p.grau) return false;
    // se chegados aqui, têm o mesmo grau
    for (int i = 3; i >= 0; i--) {
        if (coeficiente[i] > p.coeficiente[i]) return true;
        if (coeficiente[i] < p.coeficiente[i]) return false;
    }
    return false; // se chegados aqui, os polinómios são iguais
}

CPolinomio CPolinomio :: operator + (const CPolinomio &b) {
    CPolinomio s;

    for (int i=0; i<4; i++) s.coeficiente[i]=coeficiente[i]+b.coeficiente[i];
    s.grau = s.determinaGrau();
    return s;
}
```

# Programação Orientada a Objetos

- Classes de Objetos
- Construtor e Destruitor
- Ponteiros para Objetos
- Sobrecarga de Operadores
- **Membros Estáticos**

# Membros Estáticos

- Uma classe pode ter **atributos e métodos estáticos**
- **Atributo Estático**, conhecido por “variável de classe”:
  - O seu valor é partilhado por *todos* os objetos da classe
  - Exemplo: número de objetos da classe que existem
- **Método Estático**:
  - Pode ser chamado sem instanciar um objeto, indicando apenas o nome da classe e o operador de escopo `::`
  - Se declarado como `public`, permite aceder ao valor de atributos estáticos `private` ou `protected` sem necessidade de se instanciar objetos da classe
  - Apenas pode aceder a atributos estáticos. Sem acesso a `this` !

## Membros Estáticos

```

class CTeste {
private:
    static int contador;
public:
    CTeste () { contador ++; }
    ~CTeste () { contador --; }
    static int getContador(void){return contador;}
};

int CTeste::contador = 0; // inicializa o atributo estático

int main ()
{
    CTeste a, b[5], *c = new CTeste;

    cout << a.getContador() << ' ' << CTeste::getContador();
    delete c;
    cout << endl << CTeste::getContador() << endl;
}

```

7 7  
6

## Programação Orientada a Objetos

- Classes de Objetos
- Construtor e Destruitor
- Ponteiros para Objetos
- Sobrecarga de Operadores
- Membros Estáticos
- Funções e Classes Amigas

## Funções Amigas

- Sendo definidas fora do contexto da classe de que são «amigas», têm acesso aos membros `private` e `protected` desta
- Esta permissão é conferida através da palavra reservada `friend` seguida da declaração da função

```
class CRetangulo {
    private:
        int largura, altura;
    public:
        void inicializaValores(int, int);
        int area(void);
        friend CRetangulo duplica(const CRetangulo &);
};
```

## Funções Amigas

```
CRetangulo duplica(const CRetangulo &r){ // função global
    CRetangulo rTemp;

    rTemp.largura = r.largura * 2;           // acesso a atributos
    rTemp.altura = r.altura * 2;             // privados de CRetangulo
    return rTemp;
}

int main () {
    CRetangulo retA, retB;
    retA.inicializaValores(2, 3);

    retB = duplica(retA);

    cout << "A área é: " << retB.area();
}
```

A área é: 24

## Funções Amigas

- As funções amigas de uma classe são fundamentais para se implementar a sobrecarga de operadores em que o 1.º operando é uma constante, e.g. uma constante inteira...

```
int main () {
    CRetangulo retA(2, 3), retB;

    retB = 3 + retA; // 1.º operando é uma constante do tipo int!
    // o operador + definido na classe anteriormente não serve...

    cout << "A área é: " << retB.area();
}
```

## Funções Amigas

```
class CRetangulo {
private:
    int largura, altura;
public:
    CRetangulo(int, int); // construtor por enumeração
    ...
    friend CRetangulo operator+(const int, const CRetangulo &);
};

// Função amiga da classe CRetangulo:
CRetangulo operator+(const int aumento, const CRetangulo &b)
{ return CRetangulo(aumento + b.largura, aumento + b.altura); }

int main () {
    CRetangulo retA(2, 3), retB;

    retB = 3 + retA;
    cout << "A área é: " << retB.area();
}
```

A área é: 30

## Classes Amigas

- Uma classe «amiga» de uma outra classe tem acesso aos membros private e protected dessa classe de que é «amiga»
- Esta permissão é conferida através da palavra reservada friend seguida da declaração da classe

```
class CRetangulo;
class CQuadrado {
    int lado;
public:
    CQuadrado(void);
    CQuadrado(int);
    friend class CRetangulo;
};
```

## Classes Amigas

```
class CRetangulo {
    int largura, altura;
public:
    ...
    void converteQ(const CQuadrado &);

};

//-----
void CRetangulo::converteQ(const CQuadrado &a)
{ largura = altura = a.lado; } // lado é 'private'
//-----

int main () {
    CQuadrado q(4);      // quadrado de lado = 4
    CRetangulo r;
    r.converteQ(q);
    cout << "Resultado: " << r.area();
}
```

Resultado: 16

# Composição de Classes

- Uma classe pode ter como atributos objetos de outras classes
- Esses objetos são construídos antes do objeto do qual são atributos

```
class CData {  
    int dia, mes, ano;  
public:  
    CData(void); // construtor por omissão: 01/01/2000  
    CData(int di, int me, int an);  
};  
class CPessoa {  
    char nome[50];  
    CData dNasc;  
public:  
    CPessoa(void);  
    CPessoa(char *nom, int di, int me, int an);  
};
```